



Tecnicatura Universitaria en Inteligencia Artificial

**Procesamiento del Lenguaje Natural(NLP)**

Informe Trabajo Práctico Final

Alumna: Sol Kidonakis

**INFORME**

**EJERCICIO 1**

## 1. Introducción al Proyecto

El presente proyecto tiene como objetivo el desarrollo de un chatbot experto utilizando la técnica de **Retrieval-Augmented Generation (RAG)**. En el ámbito del procesamiento del lenguaje natural, un chatbot tradicional basado únicamente en modelos de lenguaje enfrenta limitaciones, especialmente cuando se le exige responder con precisión a preguntas que requieren acceso a información específica o actualizada. Aca es donde RAG ofrece una solución innovadora.

La técnica RAG combina dos procesos: la generación de texto mediante un modelo de lenguaje y la recuperación de información relevante desde una base de datos, documentos u otras fuentes de conocimiento. De esta manera, no solo se genera texto basado en patrones aprendidos, sino que se enriquece la respuesta con datos exactos recuperados en tiempo real. Esta combinación permite al chatbot ofrecer respuestas más precisas y contextualizadas, abordando las limitaciones de los sistemas puramente generativos.

**Problema a Resolver:** El desafío que se aborda en este proyecto es la necesidad de un chatbot que no solo comprenda el lenguaje natural, sino que también tenga la capacidad de acceder a información precisa y actualizada para responder preguntas detalladas. Los modelos de lenguaje tradicionales, aunque potentes, carecen de acceso directo a datos externos, lo que limita su capacidad para responder a consultas que dependen de información específica. Con RAG, buscamos superar esto.

## 2. Extracción de Texto desde Documentos PDF

Se utiliza la biblioteca PyPDF2 para la extracción de texto de archivos PDF. Este paso es crucial ya que los documentos PDF contienen la información que el chatbot utilizará para responder preguntas.

## 3. División del Texto en Fragmentos

```
splitter = RecursiveCharacterTextSplitter(chunk_size=chunk_size, chunk_overlap=chunk_overlap)
```

Se utiliza RecursiveCharacterTextSplitter de Langchain para dividir el texto en fragmentos de tamaño controlado (chunk_size=1000) con un solapamiento (chunk_overlap=200). Este enfoque facilita la indexación y recuperación de

información, asegurando que no se pierda contexto importante entre los fragmentos.

## 4. Configuración y Uso de ChromaDB

```
# Configurar ChromaDB
client = chromadb.Client()

# Crear o recuperar la colección
collection = client.get_or_create_collection("collection")
```

ChromaDB se eligió como la base de datos de vectores para almacenar los embeddings generados a partir de los textos procesados. Para realizar los embeddings utilizamos SentenceTransformer,este modelo es especialmente útil en tareas que requieren encontrar similitudes entre textos o la recuperación de información relevante. ChromaDB se elige por su capacidad de manejar grandes volúmenes de datos de manera eficiente, lo cual es fundamental para una respuesta rápida del chatbot.

Se genera un embedding para cada fragmento de texto utilizando el modelo SentenceTransformer. Estos embeddings son almacenados en ChromaDB, permitiendo una búsqueda eficiente basada en similitud de vectores.La base de datos permite manejar grandes volúmenes de datos, lo que es crucial a medida que se incrementa la cantidad de información a la que el chatbot tiene acceso.

## 5. Integración de Datos CSV

Se cargan datos desde un archivo CSV (WorldPopulation2023.csv) que contiene información relevante sobre la población. Estos datos se integran para permitir que el chatbot responda preguntas específicas relacionadas con la población mundial o regional.

## 6. Integración con Wikidata usando SPARQL

Se utiliza la API de SPARQL para realizar consultas dinámicas en Wikidata,que nos permite acceder a una base de datos de conocimiento estructurado.

## 7. Búsqueda

Las funciones de busqueda están diseñada para recuperar información relevante desde diferentes tipos de fuentes de datos: vectores en ChromaDB, datos tabulares en un CSV, y base de datos de grafos. El chatbot utiliza estas búsquedas para proporcionar respuestas informadas y contextuales basadas en la consulta específica del usuario.

## 8. Generación de Respuestas con Zephyr(modelo basado en LLM)

Para generar respuestas precisas, se implementó el modelo **Zephyr** de Hugging Face. Este modelo se utiliza para generar respuestas basadas en un few-shot prompt, donde se proporciona al modelo varios ejemplos de preguntas y respuestas antes de generar una nueva respuesta.

El modelo Zephyr se guía mediante ejemplos previos, lo que ayuda a enfocar la generación de texto en la dirección correcta.El modelo ajusta su salida según el contexto y los ejemplos proporcionados.

## 9. Clasificación usando Logistic Regression

```
# Entrenar el modelo de regresión logística
clf = LogisticRegression(max_iter=1000, random_state=42)
clf.fit(train_embeddings, train_labels)
```

Además del modelo basado en LLM, se entrena un modelo de regresión logística utilizando embeddings generados por SentenceTransformers. Este clasificador utiliza estos embeddings para clasificar las consultas del usuario en categorías como "Argentina", "Brasil" y "Población Mundial".

## 10. Comparación de Modelos

Finalmente, se compara el rendimiento del modelo LLM y el clasificador basado en embeddings en términos de precisión. Esto proporciona una visión clara de cuál de los dos enfoques es más efectivo para el contexto del chatbot.

El modelo basado en embeddings probablemente está dando mejores resultados porque está específicamente optimizado para esta tarea mediante el entrenamiento supervisado. El modelo de regresión logística está entrenado directamente con datos etiquetados, lo que significa que ha aprendido específicamente a distinguir entre las diferentes clases (población de Argentina, PIB de Brasil, etc.) basándose en las características de los embeddings.

## 11.Preprocesamiento de Textos y Cálculo de Similitud.

Utilizamos una función para eliminar los URLs, números y espacios redundantes, limpiando el texto para reducir el ruido que podría afectar la calidad de la similitud calculada.

## 12. Procesar la consulta del usuario

Utilizo TfidfVectorizer para transformar los textos en vectores y cosine similarity para calcular la similitud entre la consulta y los documentos. Esta técnica es utilizada para identificar los fragmentos de texto más relevantes.

```
def calculate_similarity(query, documents):
    processed_docs = [preprocess_text(doc) for doc in documents]
    texts = [query] + processed_docs
    vectorizer = TfidfVectorizer().fit_transform(texts)
    vectors = vectorizer.toarray()
    cosine_similarities = cosine_similarity(vectors[0:1], vectors[1:]).flatten()
```

## 13.Procesamiento de Resultados

Las funciones para procesar resultados de ChromaDB, CSV y grafos están diseñadas para extraer la información más relevante basándose en la similitud de los documentos con la consulta.

## 14. Generación de Respuestas con Plantillas y Zephyr

```
# Función para generar respuestas utilizando plantilla Jinja
def zephyr_instruct_template(messages, add_generation_prompt=True):
    template_str = "{% for message in messages %}"
    template_str += "{% if message['role'] == 'user' %}"
```

Usar estas templates nos permite la creación dinámica de prompts para el modelo Zephyr. Esto facilita la personalización de las respuestas generadas en función del contexto proporcionado.

**Preparación del Prompt para el Modelo:**

```
def prepare_prompt(query_str: str, nodes: list):
    # Concatenar todo el texto de contexto en una cadena
    context_str = '\n'.join([node.get("text", "") if isinstance(node, dict) else node for node in nodes])

    # Crear el prompt directamente con la pregunta y el contexto
    prompt_content = (
        "Contexto:\n"
        f"{context_str}\n"
        "Pregunta: {query_str}\n"
        "Respuesta: "
    ).format(query_str=query_str)
```

El prompt preparado incluye tanto la consulta del usuario como el contexto relevante extraído de las fuentes de datos, lo que permite al modelo Zephyr generar respuestas más precisas y adecuadas. Estructurar el prompt de esta manera asegura que el modelo entienda la pregunta en el contexto correcto, mejorando la calidad de la respuesta.

## 15. Función Principal del Chatbot

```
def chatbot_response(query, model, collection, csv_data):
    language = detect(query)
    classification = classify_with_llm(query)
```

Esta función integra todas las etapas del procesamiento, desde la clasificación de la consulta hasta la recuperación de información y la generación de la respuesta.

## Conclusiones

Si bien las respuestas del chatbot son en su mayoría precisas en cuanto a los datos presentados, la exhaustividad es un problema. El chatbot a menudo no logra entregar una respuesta completa, lo cual podría generar confusión o insatisfacción en el usuario.

Intenta contextualizar la información recuperada antes de proporcionar una respuesta. Sin embargo, la forma en que esta contextualización se presenta puede no ser completamente fluida para el usuario, ya que el texto recuperado a veces se mezcla de manera desorganizada con la respuesta generada.

En resumen, el chatbot tiene una base funcional sólida pero requiere optimizaciones significativas para mejorar su capacidad de generar respuestas que sean tanto precisas como coherentes desde la perspectiva del usuario.

**Enlaces a los Modelos y Librerías Utilizados**

- **SentenceTransformers:** https://www.sbert.net/
- **Langchain:** https://www.langchain.com/
- **ChromaDB:** https://www.chromadb.com/
- **Zephyr:** https://huggingface.co/HuggingFaceH4/zephyr-7b-beta
- **spaCy:** https://spacy.io/
- **Wikidata SPARQL Endpoint:** https://query.wikidata.org/

## Ejercicio 2

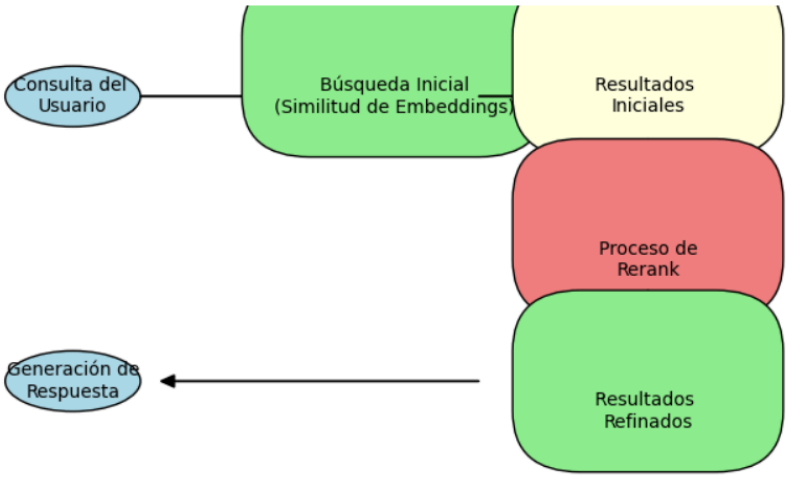
**1) Explicación del Concepto de Rerank y su Impacto en el Desempeño**

**Rerank** es un proceso crítico en el campo del procesamiento del lenguaje natural (NLP) y la recuperación de información. En el contexto de **Retrieval-Augmented Generation (RAG)**, **Rerank** se refiere a la etapa en la que los resultados de una búsqueda inicial (normalmente basada en la similitud de embeddings) se reordenan según su relevancia para la consulta del usuario. Este reordenamiento se realiza utilizando modelos más sofisticados o criterios adicionales que pueden considerar factores como el contexto, la semántica profunda o la estructura de la información.

**Conceptos Fundamentales:**

1. **Ranking Inicial:**
    - Como describen Burges et al. (2014) en su trabajo sobre redes neuronales profundas para clasificación, el primer paso en la mayoría de los sistemas de recuperación es generar un conjunto de resultados candidatos utilizando un enfoque rápido y eficiente, como la similitud de vectores.
2. **Rerank:**
    - Después de la recuperación inicial, se aplica un modelo más complejo (como un modelo de lenguaje profundo, por ejemplo, BERT) para evaluar con mayor precisión la relevancia de cada uno de los resultados candidatos. Zamani y Craswell (2020) explican que este proceso de Rerank permite al sistema priorizar resultados que no solo son similares en términos de embeddings, sino que también son semánticamente más relevantes para la consulta específica del usuario.
3. **Impacto en el Desempeño:**
    - El proceso de Rerank tiene un impacto directo en la precisión y relevancia de las respuestas generadas por el sistema. Al reordenar los resultados, se mejora la calidad de la información que se utiliza para generar la respuesta final. Esto es particularmente importante en sistemas como RAG, donde la calidad de los fragmentos de información recuperados determina en gran medida la utilidad de la respuesta generada.

**Diagrama:**



## 2) Aplicación de Rerank en el Código

```python
# Procesar resultados de ChromaDB
def process_chromadb_results(results, query):
    if not results or 'documents' not in results:
        return "No se encontraron documentos relevantes."
    documents = results['documents'][0]
    most_relevant_document = calculate_similarity(query, documents)
    return most_relevant_document

# Procesar resultados de CSV
def process_csv_results(query, df):
    if df.empty:
        return "No se encontraron datos relevantes en el CSV."
    documents = df.apply(lambda row: ' '.join(row.astype(str)), axis=1).tolist()
    most_relevant_document = calculate_similarity(query, documents)
    return most_relevant_document[:500]

# Procesar resultados de Grafos
def process_graph_results(query, results):
    if not results:
        return "No se encontraron datos relevantes en la base de datos de grafos."
    documents = [f"{result['subject']} - {result['predicate']} - {result['object']}" for result in results]
    most_relevant_document = calculate_similarity(query, documents)
    return most_relevant_document[:500]
```

En este caso,podemos ver como ya se está aplicando una técnica de reordenamiento basada en la similitud de coseno para refinar los resultados obtenidos en la búsqueda inicial.

Después de que se obtiene un conjunto de documentos que potencialmente responden a la consulta del usuario, se calcula la similitud de coseno entre la consulta y cada uno de los documentos. Los documentos se reordenan en función de esta similitud, priorizando aquellos que son más similares al contexto de la consulta.
Aunque la similitud de coseno es una técnica efectiva, se puede considerar aplicar un proceso de Rerank más sofisticado en lugar de usar solo la similitud de coseno,o aplicar un modelo de lenguaje profundo, como BERT, para evaluar la relevancia semántica de los documentos. Este modelo podría reordenar los resultados en función de una comprensión más profunda del contexto y el significado.

**Fuentes de Información Utilizadas:**


1. **Burges, C., Shaked, T., Renshaw, E., Deeds, M., Hamilton, N., Hullender, G. (2014).** "Learning to Rank with Deep Neural Networks."
2. **Zamani, H., Craswell, N. (2020).** "Ranking and Reranking with Deep Learning."
3. **Hugging Face Documentation:** "RAG: Retrieval-Augmented Generation." Hugging Face Blog